Understood. Let's restructure the project to have a main module named `command_processor.py` in the `Brain/services` folder and a subfolder `command_processor` that contains the functions broken down into separate files. This approach will enhance modularity and readability.

Updated Project Structure

```
jarvis/
├── BOS/
│   ├── __init__.py
│   ├── listen.py
│   ├── process.py
│   ├── speak.py
├── Brain/
│   ├── __init__.py
│   ├── app.py
│   ├── config.py
│   ├── intents.json
│   ├── services/
│   │   ├── __init__.py
│   │   ├── command_processor.py
│   │   ├── command_processor/
│   │   │   ├── __init__.py
│   │   │   ├── classify_command.py
│   │   │   ├── extract_arguments.py
│   │   │   ├── execute_function.py
│   │   ├── response.py
│   │   ├── memory.py
│   │   ├── news.py
│   │   ├── weather.py
├── jarvis.py
├── setup.py
├── requirements.txt
```

Code for Each File

`BOS/__init__.py`

```
# Empty file to make BOS a Python package
```

`BOS/listen.py`

```python
import speech_recognition as sr

def listen():
    recognizer = sr.Recognizer()
    with sr.Microphone() as source:
        print("Listening...")
        audio = recognizer.listen(source)
        try:
            command = recognizer.recognize_google(audio)
            print(f"User said: {command}")
            return command
        except sr.UnknownValueError:
            print("Sorry, I did not understand that.")
            return None
        except sr.RequestError:
            print("Sorry, my speech service is down.")
            return None
```

`BOS/process.py`

```python
import requests

def process(command):
    response = requests.post("http://localhost:5000/api/process", json={"command": comma
    return response.json()
```

`BOS/speak.py`

```python
import pyttsx3

def speak(text):
    engine = pyttsx3.init()
    engine.say(text)
    engine.runAndWait()
```

`Brain/__init__.py`

```
# Empty file to make Brain a Python package
```

`Brain/app.py`

```
from flask import Flask
from .services.command_processor import command_processor_bp

app = Flask(__name__)

app.register_blueprint(command_processor_bp, url_prefix='/api')

if __name__ == "__main__":
    app.run(port=5000, debug=True)
```

`Brain/config.py`

```
# Configuration settings
DATABASE_URI = 'mongodb://localhost:27017/jarvis'
```

`Brain/intents.json`

```
{
  "intents": [
    {
      "tag": "news",
      "patterns": ["tell me news", "what's trending", "latest news", "trending news", "y
      "function": "get_news",
      "arguments": [
        {
          "name": "time",
          "types": ["DATE", "RB"]
        },
        {
          "name": "category",
          "types": ["NN", "NNS", "EVENT", "ORG"]
        }
      ]
    },
    {
      "tag": "weather",
      "patterns": ["what's the weather", "weather today", "current weather"],
      "function": "get_weather",
      "arguments": [
        {
          "name": "time",
          "types": ["DATE", "RB"]
        },
        {
          "name": "location",
          "types": ["GPE"]
        }
      ]
    },
    {
      "tag": "memory",
      "patterns": ["remember", "recall", "remind me"],
      "function": "handle_memory",
      "arguments": [
        {
          "name": "task",
          "types": ["NN", "NNS", "VB"]
        }
      ]
    },
    {
      "tag": "response",
      "patterns": ["hello", "how are you", "tell me a joke"],
      "function": "get_response",
      "arguments": []
    }
  ]
}
```

`Brain/services/__init__.py`

```
# Empty file to make services a Python package
```

`Brain/services/command_processor.py`

```
import json
from flask import Blueprint, request, jsonify
from .command_processor.classify_command import classify_command
from .command_processor.execute_function import execute_function

command_processor_bp = Blueprint('command_processor', __name__)

with open('Brain/intents.json') as file:
    intents = json.load(file)["intents"]

@command_processor_bp.route('/process', methods=['POST'])
def process_command():
    command = request.json.get('command')
    intent, arguments = classify_command(command, intents)
    response = execute_function(intent, arguments)
    return jsonify(response), 200
```

`Brain/services/command_processor/__init__.py`

```
# Empty file to make command_processor a Python package
```

`Brain/services/command_processor/classify_command.py`

```
from .nlp_processing import NLPProcessor

nlp_processor = NLPProcessor()

def classify_command(command, intents):
    intent_scores = {}
    for intent in intents:
        intent_scores[intent["tag"]] = 0

        # Check for direct pattern match
        for pattern in intent["patterns"]:
            if pattern in command.lower():
                intent_scores[intent["tag"]] += 1

        # Perform NLP processing for additional context
        tokens = nlp_processor.tokenize(command)
        pos_tags = nlp_processor.pos_tagging(tokens)
        entities = nlp_processor.ner(command)

        # Adjust scores based on POS tags and named entities
        for token, tag in pos_tags:
            if tag.startswith("VB"):  # Verbs
```

```
                intent_scores[intent["tag"]] += 0.5
            elif tag.startswith("NN"):  # Nouns
                intent_scores[intent["tag"]] += 0.3

        for entity, label in entities:
            if label == "PERSON" or label == "ORG":  # Person or organization names
                intent_scores[intent["tag"]] += 1

        # Log the scores for debugging
        print(f"Scores for intent '{intent['tag']}' after processing command '{command}'

    # Choose the intent with the highest score
    max_intent = max(intent_scores, key=intent_scores.get)
    if intent_scores[max_intent] > 0:
        arguments = extract_arguments(command, max_intent, tokens, pos_tags, entities, i
        return max_intent, arguments
    else:
        return "unknown", {}

from .extract_arguments import extract_arguments
```

`Brain/services/command_processor/extract_arguments.py`

```
def extract_arguments(command, intent_tag, tokens, pos_tags, entities, intents):
    arguments = {}
    intent = next(intent for intent in intents if intent["tag"] == intent_tag)

    # Generalize argument extraction based on defined patterns
    for arg in intent.get("arguments", []):
        arg_name = arg["name"]
        arg_types = arg["types"]

        # Extract using POS tags
        for token, tag in pos_tags:
            if tag in arg_types:
                arguments[arg_name] = token.lower()

        # Extract using NER
        for entity, label in entities:
            if label in arg_types:
                arguments[arg_name] = entity.lower()

    return arguments
```

`Brain/services/command_processor/execute_function.py`

```python
from ..news import get_news
from ..weather import get_weather
from ..memory import remember, recall
from ..response import get_response

def execute_function(function_name, arguments):
    if function_name == "get_news":
        return get_news(arguments)
    elif function_name == "get_weather":
        return get_weather(arguments)
    elif function_name == "handle_memory":
        if "remember" in arguments:
            return remember(arguments)
        else:
            return recall()
    elif function_name == "get_response":
        return get_response(arguments)
    else:
        return {"response": "I'm not sure how to respond to that."}
```

`Brain/services/nlp_processing.py`

```
import spacy

class NLPProcessor:
    def __init__(self):
        self.nlp = spacy.load('en_core_web_sm')

    def tokenize(self, text):
        doc = self.nlp(text)
        return [token.text for token in doc]

    def pos_tagging(self, tokens):
        doc = self.nlp(" ".join(tokens))
        return [(token.text, token.tag_) for token in doc]

    def ner(self, text):
        doc = self.nlp(text)
        return [(ent.text, ent.label_) for ent in doc.ents]
```

`Brain/services/response.py`

```
def get_response(command):
    responses = {
        "hello": "Hello! How can I assist you today?",
        "how are you": "I'm just a program, but I'm functioning as expected!",
        "tell me a joke": "Why did the scarecrow win an award? Because he was outstandin
    }
    for key

 in responses:
        if key in command.lower():
            return {"response": responses[key]}
    return {"response": "I'm not sure how to respond to that."}
```

`Brain/services/news.py`

```
def get_news(arguments):
    # Dummy implementation for news retrieval
    news_items = [
        "News Item 1",
        "News Item 2",
        "News Item 3"
    ]

    if "time" in arguments and arguments["time"] == "yesterday":
        news_items = ["Yesterday's News Item 1", "Yesterday's News Item 2"]

    if "category" in arguments and arguments["category"] == "cricket":
        news_items = ["Cricket News Item 1", "Cricket News Item 2"]

    return {"response": news_items}
```

`Brain/services/weather.py`

```
def get_weather(arguments):
    # Dummy implementation for weather retrieval
    location = arguments.get("location", "your location")
    time = arguments.get("time", "now")

    weather_info = {
        "location": location,
        "time": time,
        "forecast": "sunny"
    }

    return {"response": weather_info}
```

`setup.py`

```
import os

def install_dependencies():
    os.system("pip install -r requirements.txt")

def setup_configuration():
    # Additional setup tasks like asking user arguments can be implemented here
    pass

if __name__ == "__main__":
    install_dependencies()
    setup_configuration()
    print("Setup complete.")
```

`requirements.txt`

```
Flask
speechrecognition
pyttsx3
requests
spacy
en_core_web_sm
```

Instructions to Set Up and Run

1. **Install Dependencies**:

```
   python setup.py
```

2. **Run Flask Server**:

```
python Brain/app.py
```

3. **Start Jarvis**:

```
python jarvis.py
```

This structure should provide you with a clean, modular, and maintainable codebase for your Jarvis AI assistant. Each component is designed to handle specific tasks, making it easier to manage and extend the functionality in the future

Additional servoces.

Brain/services/remind.py

```
import json
from datetime import datetime

def set_reminder_and_write_to_file(arguments):
    reminder_time = arguments['Time']
    reminder_message = arguments['Message']

    if reminder_time is not None and reminder_message is not None:
        reminder_data = {
            "reminder_time": str(reminder_time),
            "reminder_message": reminder_message
        }

        with open('reminder.json', 'w') as file:
            json.dump(reminder_data, file)

        return {"response": f"Reminder set for {reminder_time}: {reminder_message}"}

    return {"response": "Failed to set reminder. Please provide a valid time and message

def start_remind():
    try:
        with open('reminder.json', 'r+') as file:
            reminder_data = json.load(file)
            reminder_time = reminder_data.get('reminder_time')
            reminder_message = reminder_data.get('reminder_message')

            if reminder_time and reminder_time != 'next':
                reminder_datetime = datetime.strptime(reminder_time, '%Y-%m-%d %H:%M:%S'
                if reminder_datetime <= datetime.now():
                    print(f"Reminder: {reminder_message}")
                    file.seek(0)
                    file.truncate()
                    json.dump({}, file)

            elif reminder_time == 'next':
                print(f"Reminder: {reminder_message}")
                file.seek(0)
                file.truncate()
                json.dump({}, file)

    except FileNotFoundError:
        print("No reminder found!")

# Example usage:
# reminder_text = "remind me on 24 January at 6pm to make index.html"
# reminder_obj = get_remind(reminder_text)
# set_reminder_and_write_to_file(reminder_obj)
```

Brain/services/bright.py

```
Copy code
import ctypes

def change_brightness(arguments):
    action = arguments.get("action","decrease")
    brightness = ctypes.c_ulong()
    NoneType = type(None)

    if ctypes.windll.powrprof.PowerGetActiveScheme(None, ctypes.byref(guid)) and \
       ctypes.windll.powrprof.PowerReadACValue(None, ctypes.byref(guid), None, None, cty
       isinstance(brightness.value, (int, NoneType)):
        brightness_level = brightness.value

        if action == "increase":
            new_brightness = int(brightness_level * 1.1)
            if new_brightness <= 100:
                ctypes.windll.powrprof.PowerWriteACValueIndex(None, ctypes.byref(guid), 
                return {"response": "Brightness increased."}
            else:
                return {"response": "Maximum brightness reached."}

        elif action == "decrease":
            new_brightness = int(brightness_level * 0.9)
            if new_brightness >= 0:
                ctypes.windll.powrprof.PowerWriteACValueIndex(None, ctypes.byref(guid), 
                return {"response": "Brightness decreased."}
            else:
                return {"response": "Minimum brightness reached."}
    return {"response": "Failed to change brightness."}
```

Brain/services/win.py

```
import pygetwindow as gw

def perform_window_action(arguments):
    action = arguments.get("action")
    window_name = arguments.get("window")

    if action and window_name:
        windows = gw.getAllWindows()
        target_window = next((window for window in windows if window_name in window.titl

        if target_window:
            if action == "maximize":
                target_window.maximize()
                return {"response": f"Maximized {window_name} window."}
            elif action == "minimize":
                target_window.minimize()
                return {"response": f"Minimized {window_name} window."}
            elif action == "info":
                return {"response": f"Window info: {target_window}"}
        return {"response": f"{window_name} window not found."}

    return {"response": "Action or window name not specified."}
```